DNSC 6279 - Data Mining - Project Report

Daniel Chudnov

1 Problem Statement

Github (github.com) is a popular social network for software developers and the code they produce. It allows individual programmers and groups or organizations to publish and share their code using the Git version control software application, and it enables powerful features for sharing changes to code. Many of the world's most widely used software products are shared on Github, such as the Linux kernel, and even projects that aren't actively developed using Github mirror their code to Github for the benefits of visiblity and ease of use Github offers, including the Apache web server, the Python programming language, the PostgreSQL database, and Git itself. By a rough count (using the monotonically-increasing "id" variable present in API calls to list repositories registered in Github, more about which below), there have been well over nineteen million software project repositories registered in Github since its inception in early 2008. In my own work at GW and within my industry (software development in libraries) I and most of my colleagues perform the majority of our work using Github, so I have a strong personal interest in its success as a platform and what we may learn from it.

One of the hardest choicest to make as a software developer – especially as one who works with free/libre/open source software often, as I and my colleagues do – is which software projects and products written by other people we should use as a basis upon which to build our own software. For any common computing problem there are at least three different packages available in any programming language of choice, so which is the best to use? Which packages work best, and which new initiatives are likely to "get traction"? There is a natural tension between doing new work in software and building that new work upon other, existing, reliable packages, to narrow the scope of innovation and to allow for a stable development environment. Because of this software developers have a natural tendency to look for certain things when evaluating a package: Who is its author? How long has it been under development, and how active has its development been lately? Does the community use tickets and milestones to plan releases? Have others "forked" the package (made their own copy) and submitted their own changes to it? No one answer to these questions is a perfect piece of information, rather we go on intuition and experience to make these choices.

The Github API (developer.github.com/v3/) provides an opportunity to examine these choices, and to determine whether we can predict the potential success of a software project using data rather than simply by intuition. For each repository in Github, extensive metadata about that repository and its activity is available through a well-documented set of API calls. For this project, I collected this metadata for roughly 10,000 repositories registered between 2008-2010, and I attempted to build several models that might tell us more about what we can determine automatically about any repository in Github, based on how the project is managed in Github, the programming language choices its authors make, and patterns of recent development activity for each project. Although I do not have a great deal of confidence in these models, the process has taught me a lot about the data mining process, especially the critical data collection and pre-processing steps, and also about Github as a community.

2 Data Selection

I used the following calls to Github's API to return data for each repository. Note that each URL begins with "https://api.github.com". These API calls are documented extensively on the Github API site.

Path	Description
/repositories	A pageable list of repositories, starting from oldest
/repos/owner/name	Basic metadata (owner, name from previous call)
/repos/owner/name/contributors	Pageable details about all code contributors
/repos/owner/name/stats/participation	Counts of commits by owner vs. all others, past year
/repos/owner/name/languages	Relative size of code by programming language
/repos/owner/name/stats/code_frequency	Counts of lines added/removed weekly, past year

These calls generated a lot of raw data for each repository which I then processed further, as described in Section 3 below. For each repository, basic metadata of value for this project includes:

Field	Description	Type
id	unique identifier	int
owner	repository owning user	string
name	repository name	string
created_at	date of repository creation	date
pushed_at	date of last "pushed" commit (change)	date
fork	whether this is a "fork" or copy of another repository	boolean
forks_count	number of forks of this repository	int
has_wiki	whether the project uses Github's wiki pages	boolean
has_issues	whether the project uses Github's issue tracking	boolean
has_downloads	whether the project has downloadable package builds	boolean
parent_id	id of project from which this project was forked, if applicable	int
source_id	id of original project from which this and parents were forked	int
size	size of the repository, in source lines of code	int
open_issues_count	number of currently-open issues in Github's issue tracker	int
stargazers_count	# users who have "starred" (or favorited) this repository	int
subscribers_count	# users who subscribe to update notifications	int
network_count	number of all forks in network of source repository	int

For this project, I used "id" as an identifier variable, "stargazers_count" as the target variable, and rejected "owner" and "name" although I carried them through into SAS as useful metadata to check values against (we can easily construct a Github URL to a repository if we have its owner and name).

Further details about processed variables are specified in the next section.

Before continuing, however, a few details about the process of fetching data from the API are worth noting. Collecting the data from Github proved to be more of a challenge than I expected for several reasons:

- Github limits API calls to 5000/hour, so at the rate of six calls per repository, it took several overnight runs to collect full data for over 10,000 repositories (and re-runs after adjusting the code several times).

- The fetch script I wrote honors this limit by attempting to wait between calls according to the amount of time remaining before the wait limit is reset by the API, but I found that this reset time was sliding backwards, causing my code to issue calls after a longer delay than necessary. After testing this very carefully I wrote to Github staff about it, and reported it as a bug with details about my queries. They acknowledged it was a bug in their system and thanked me for the report, but as far as I know the problem has not yet been fixed. I estimate this slowed down my data collection by 30-40%.

- Some of the API calls (specifically the ones that include '/stats' in the URL pattern) are documented as "slow" on the API side as they calculate summary statistics. These calls might result in an HTTP response code 202 Accepted (rather than 200 OK) which indicates that the request has been acknowledged but the response is not yet ready. To accomodate this

possibility, which I saw regularly as my fetch script executed, I implemented a simple linear backoff strategy to enable subsequent calls with an increasing delay between each call. This strategy worked, but it also slowed down the data fetching process.

- Some registered repositories have no meaningful data within them; I chose to ignore these, and did not include them in the sample.
- Some API calls (e.g. /stats/contributors) require paging to fetch all results; I used /contributors instead to save time, but only after first fetching the more complete details, which did not prove interesting after all.
- Sometimes my script simply failed (due to bad logic or improper error handling in my code, failed network connection, etc.). At first, I had to restart it over from scratch, but after losing data a few times I wrote an "append" feature that attempts to discover where the process last left off by checking the id of the last repository in the most recently saved file (the fetch scripts saves data in chunks of 100 repositories per file).
- After working with the results of metadata about several thousand repositories, I noticed an obvious trend I should have considered sooner. The API call to list all the repositories in Github started back at the beginning of Github, and proceeded chronological order. Because of this, I was only retrieving data about the very oldest data in Github. This seemed like a strong bias to use as the community of users for Github started small, like for any social network, and grew first from a set of users who had particular interests in the same languages and tools the Github developers themselves used. Indeed the most active early users were the founders of Github themselves, and some of the earliest repositories were Ruby-based web applications (Github is a Ruby language application) and Ruby-based libraries for working with Git repositories. Rather than proceed with this bias, I added a "striping" function to the fetch script to jump ahead in time between calls. With the response to the query returning 100 repositories, I modified the next call to the API to jump ahead 9900 repositories, thus taking a sample of 100 out of each 10,000 in chronological time, or 1% of repositories overall. Without this change, my sample set would have only contained the oldest repositories from the first few months of Github's existence; instead, it includes a sampling of repositories from the first three years. This still exhibits a bias of sorts as Github's largest growth has come in the past 2-3 years, but this still seemed to be a good enough sample for this project. Perhaps I will follow up with exploring a larger sample including more recent repositories in the future.

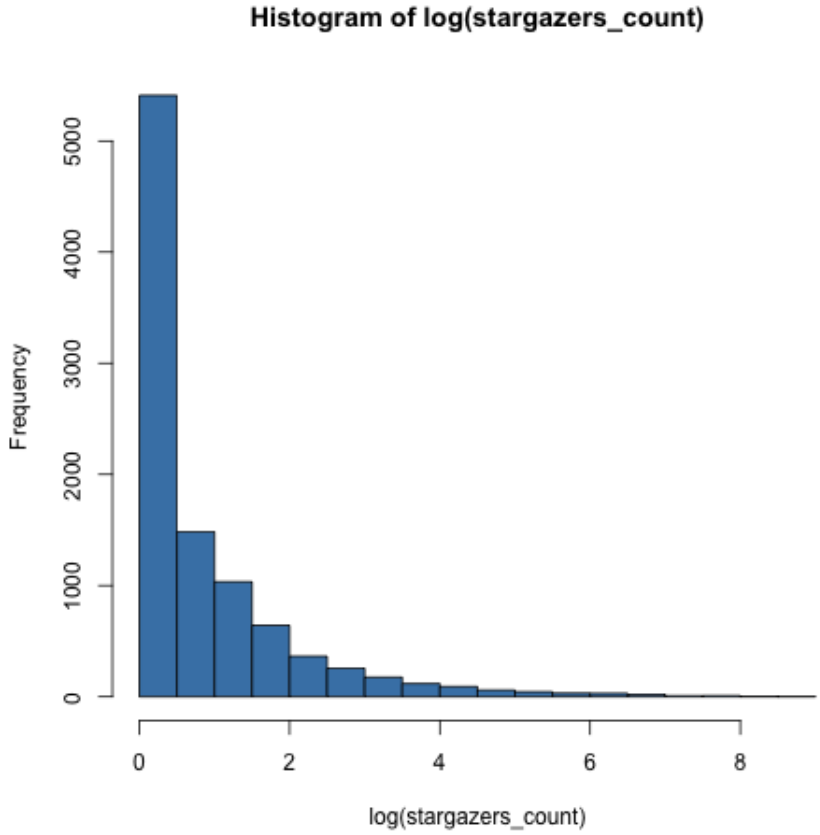
This set of issues led to a longer period of iteration over the data fetching process, delaying the processing and modeling steps. It reinforced a key lesson, that there is no such thing as "clean data", even if it's coming from a "clean API". Every API has its own quirks, and possibly flaws, and it will always take time to account for and adjust to these issues.

3 Data Pre-Processing

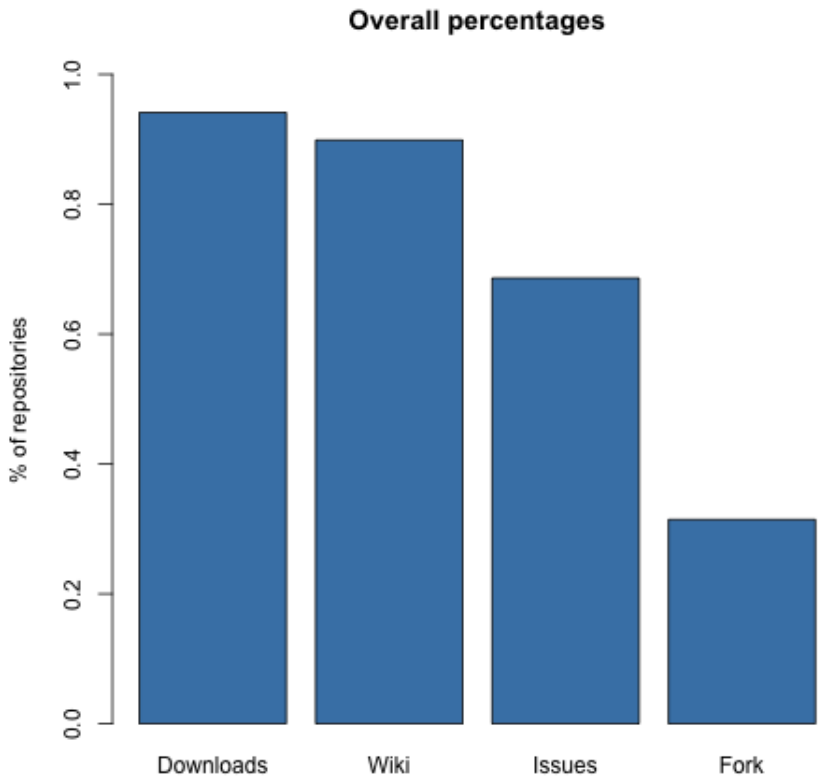
What operations did you perform on your data to organize it in a form suitable for Enterprise Miner. Specify any queries, joins, transformations, how you treat missing values etc.

I collected the data from the Github API into a set of JSON files, 100 records per file, for a total of n=10,096 repositories. For each repository record, I started with the basic JSON-based response to the basic metadata request for each repository and added results from the additional calls to the in-memory dictionary for each respository before writing each to local disk. JSON is an excellent format for this kind of work; it has all the hierarchical expressiveness of XML but very little overhead, and it takes exactly one line to either read it in or write it out from most languages.

The overall set of data I chose included a wide variety of potential values. It seemed best to explore it some and determine how best to pre-process one or more files for import into SAS prior to model building. Here are some of the basic impressions the data gave at first glance.



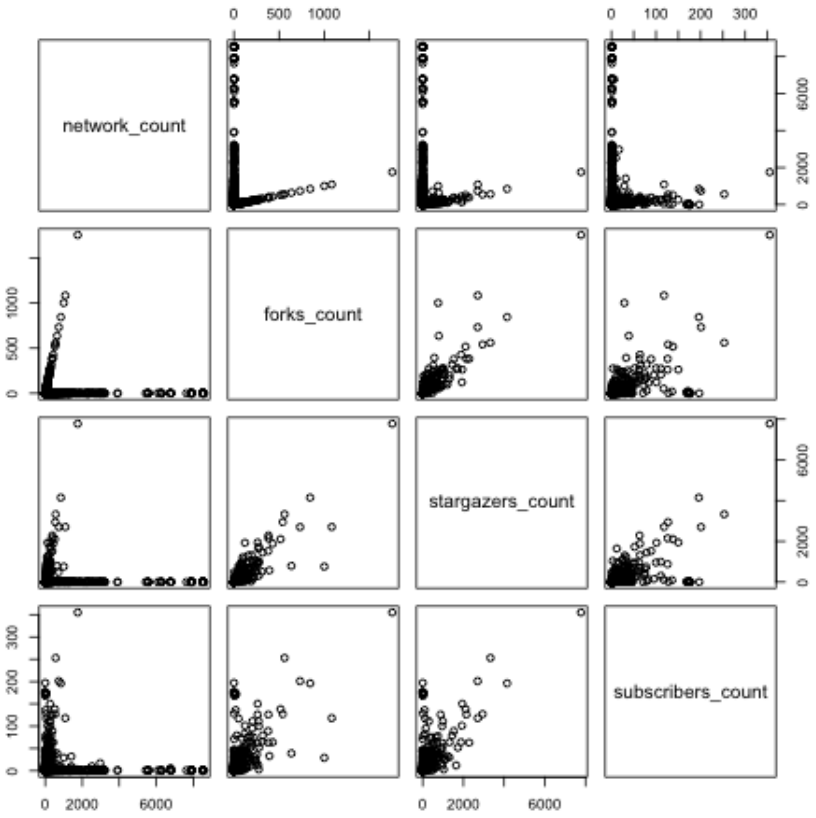
Starting with the dependent/target variable "stargazers_count", we see that it has a lognormal distribution evidenced by the histogram of the log of this value. A small proportion of repositories among the over ten thousand I collected ever "caught on".



I suspected that projects that made use of Github's issue tracking feature might be more popular because this indicates that the community of developers is actively working on the code and soliciting feedback. Similarly, but to a lesser extent, I suspected that use of other Github features such as their "file downloads" feature and project wiki pages might also indicate project health. The numbers for Downloads and Wiki seem unusually high to me, though, and although I've verified that this is what the data I fetched indicates, this does not seem to correspond to what I've seen in project sites, in particular it seems like far fewer than 90% of projects use the downloads feature, but this indicates the percentage is higher than 90%. Perhaps these values are not trustworthy. The lower number for Issues is perhaps more likely to be accurate, and the final bar for Fork, indicating that roughly 30% of repositories are forks of other repositories, seems plausible enough, although I am unexcited about its potential

predictive value.

Another exploratory chart indicates a strong relationship between forks and stargazers:



Here we can see that there is indeed a strong correlation between the number of stargazers and the number of forks of a project. This makes intuitive sense; more popular projects seem more likely to find other users working with the code, offering up changes, etc.

At this point it seemed clear that there might not be any groundbreaking paradigm to discover here; a log/power tail of projects are popular, popular projects get forked a lot, and some of the data from the API might not be 100% reliable (based on the percentages of repositories that use downloads and wikis). It seemed like a good idea to start to derive new attributes from the existing data.

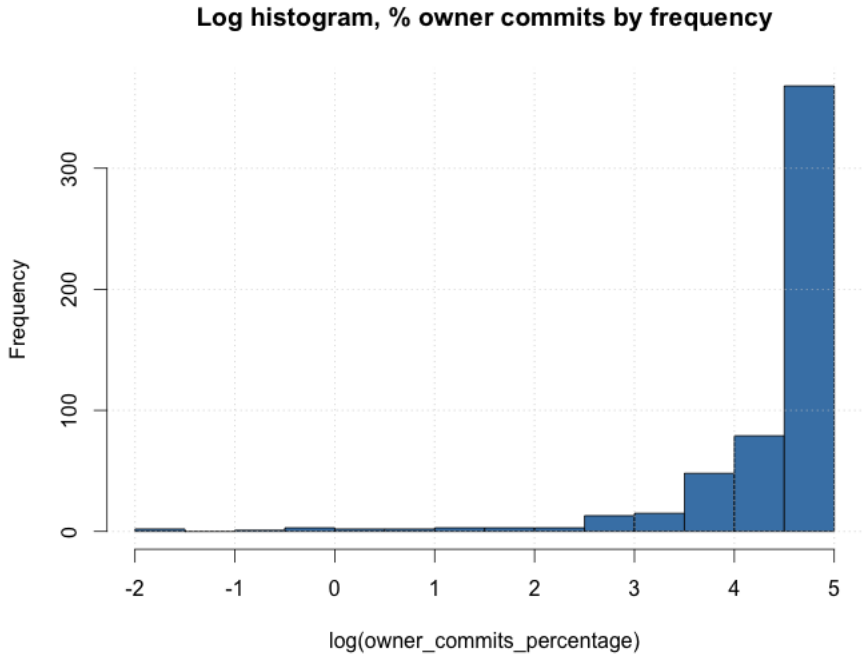
I decided to break the data down along three lines: basic attributes, code activity, and languages. The basic attributes are those already defined and listed in the previous section. To this I added the following set of computed values based on activity on the repository:

'star10', 'lang0_prop', 'lang0', 'lang1', 'lang2',

Field	Description
num_contributors	count of distinct users contributing code
num_weeks	weeks since project inception
num_weeks_since_change	weeks since last commit
lines_added	total lines added, past year
lines_added_per_week	avg lines added/week, past year
lines_subtracted	total lines subtracted, past year
lines_subtracted_per_week	avg lines subtracted/week, past year
all_commits	total count of commits
owner_commits	commits by owner, past year
owner_commits_percentage	percent of commits by owner, past year
mean_commits_per_week	mean commits/week, past year
std_commits_per_week	standard dev. of commits/week, past year

These values were readily obtained and computed from the API data provided, implemented in the script "process.py" (see Appendix).

As an example of these attributes, below is a chart of owner_commits_percentage, indicating how many changes to each repository in the past year had been commited specifically by the repository owner themself:



The vast majority of projects active in the past year had a majority of its commits in the past year created by the repository owner (the two bins at the far right with log(owner_commits_percentage $>$ four). This indicates that a

very small minority of projects have a plurality of active committers.

The last area of repository composition I reviewed was the choice of languages used in each project. I was interested to see if there was any pattern present across several years of Github data toward a preference for particular languages, or whether these matched language popularity in the wider community. Summarizing the languages used in repositories by the top two per each (more than one kind of language can be used in a single project, and often is) revealed the following basic data:

ActionScript	62
C	414
C#	137
C++	264
Clojure	49
Common Lisp	36
CSS	32
Emacs Lisp	92
Erlang	61
Go	24
Haskell	66
Java	475
JavaScript	960
Lua	42
Objective-C	159
Perl	485
PHP	549
Python	833
Ruby	2136
Scala	76
Shell	129
VimL	266

Note that Ruby is the most popular primary language, followed by JavaScript, Python, Java, PHP, Perl, and C. This is not a surprising list when compared to industry measures of the same information (for example, see http://www.tiobe.com/index.php/tiobe_index) and accounting for the early prevalence of Ruby code in Github's early days.

The most-used secondary languages are slightly more interesting:

ActionScript	15
ASP	16
C	295
C#	42
C++	142
Clojure	11
CoffeeScript	13
CSS	80
Emacs Lisp	16
Groovy	16
Java	72
JavaScript	945
Lua	15
Objective-C	52
Perl	129
PHP	124
Python	197
Ruby	317
Shell	388
VimL	29

In this list the most popular secondary languages in projects was JavaScript, by far, followed by Shell, Ruby, C, Python, and C++. This makes an intuitive sense as many projects in Github are web and mobile applications, and many of these applications use at least some JavaScript, whether the larger application itself is written in Ruby, Python, PHP, or any other language. Shell is a common choice for tool scripts around the main code of a project, with Unix commands written in Shell to automate tasks related to application configuration, data management, backups, and other maintenance tasks.

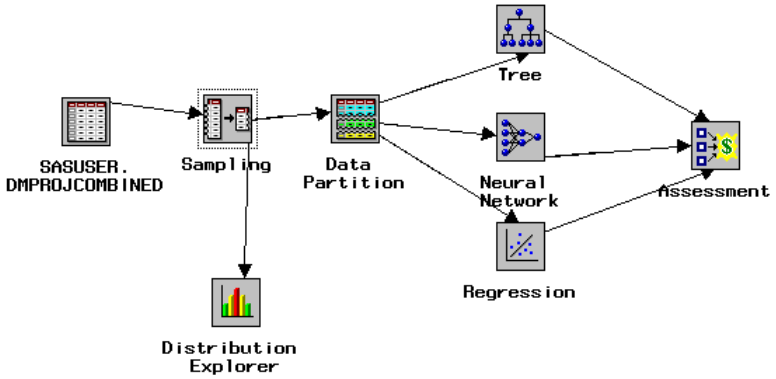
One final attribute I computed was "star10", a binary classification of whether a repository had at least ten stargazers. This proved to be useful in generating stratified samples as well as being a target variable for model generation, as described in the next section.

4 Mining Technique Selection

I chose to work with a Regression, Neural Network, and a Classification Tree for this model, as each can be used to create predictive models. At first I worked with one combined dataset and used stargazers_count as the target variable. This resulted in unexciting models that did not reveal much, largely due to the log distribution of popularity present in stargazers_count, as shown above in the histogram of log(stargazers_count).

I consulted with Professor Prasad who encouraged me to take a stratified sample to capture the "rare event" of popularity according to stargazers_count. I added a Sampling node in SAS and found that it was easiest to generate an

additional binary attribute "star10", based on whether a repository had at least 10 stargazers, to use as the stratification attribute. I created a 2000-item sample using a 50/50 equal sized stratification in slotted that between the Data Source and Partition nodes, using an 80%/20% training/validation split. I then fed that to one each of Tree, Neural Network, and Regression models, finally drawing each of those into an Assessment node. The diagram in Enterprise Miner is the following:



This represents the final model I will describe in detail now. One quick note, however: before combining all the data into one data source, I attempted to separate out the three categories of attributes mentioned above into their own datasources for basic metadata, project activity, and language choices, running the three models paradigms separately on each. This did not prove fruitful, save for one mildly interesting finding. Upon running the neural network on language choice data, it drew out "lang1 = JavaScript" as a modest predictor of project popularity. As noted in the previous section, JavaScript is a prevalent secondary language in repositories of many languages, in particular for web and mobile applications. This is not particularly meaningful in the large picture beyond keeping in mind that "web and mobile applications are popular".

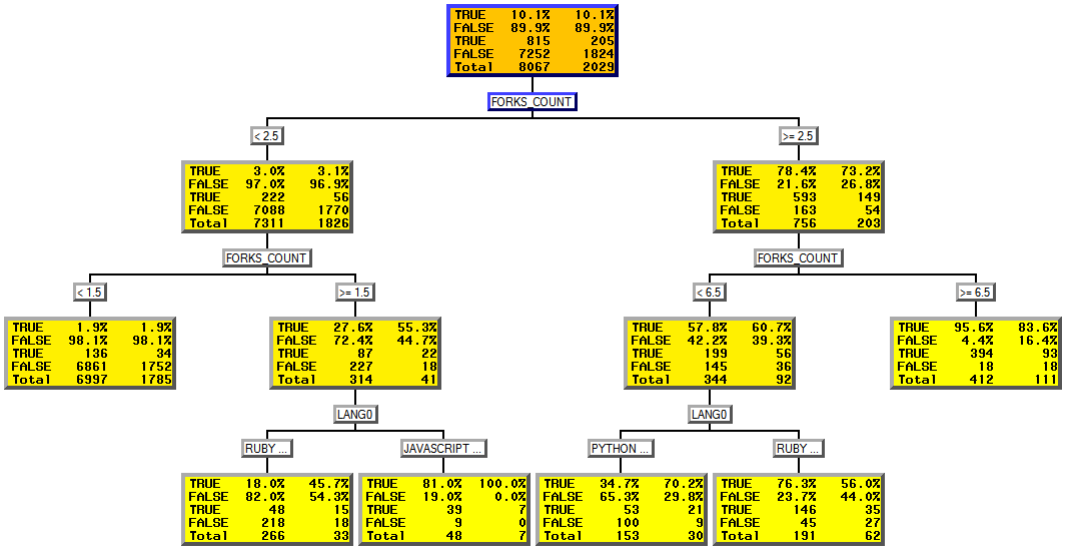
For all of these models, I began using the "stargazers_count", an interval attribute, as the target/dependent variable. This provide largely difficult to predict well, and none of the models were able to come close to a reliable prediction. Having derived the new attribute "star10", a binary classification of whether a repository had at least 10 stargazers, to use as a parameter to stratify the sample upon, I tried using star10 as the target instead of stargazers_count, thinking it might simplify the problem somewhat. After all, whether a project has 12 or 22 stargazers or whether it has 500 or 600 doesn't seem terribly important; that it has at least 10 indicates some basic level of user interest. This proved to be an improvement, as the models all generated somewhat better results, if not still overwhelmingly so.

In all of the following discussion, "star10" is the target variable.

5 Data Mining Results

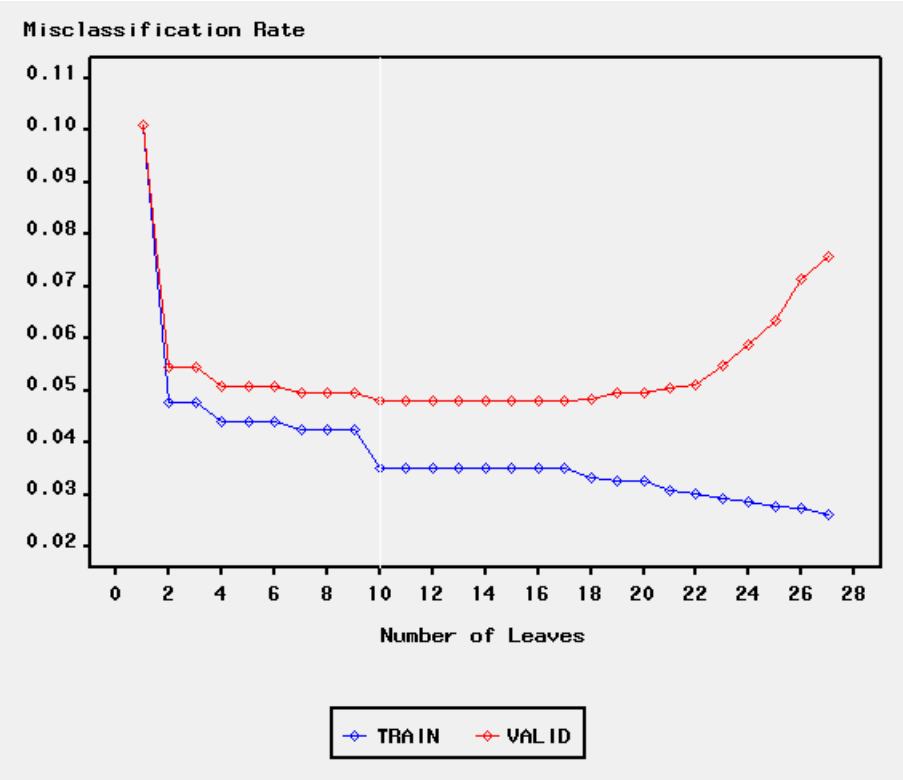
5.1 Classification Tree

For the Tree model, I used the Gini reduction splitting criterion and allowed SAS to select the model assessment measure by setting it to "automatic", and left other settings to their defaults. It produced a tree that focused primarily on the forks_count attribute, noted above as a value apparently linearly correlated with project popularity. Below is the tree:



This tree follows simple rules: if there are more than two forks, the project is typically at least modestly successful; if there are less than two, it probably isn't. At both of these ends of the spectrum, the validation data results are quite close to the training results. In the case of zero or one fork, the model correctly predicted roughly 1.9% of items as "popular"; for 3-6 forks, the validation rate of 60.7% is close to the training rate of 57.8%, and for more than seven forks, from 95.6% (training) to 83.6% (validation) isn't terrible. In between these values, particularly for repositories with exactly two forks, the validation results are poor. The lowest-level split on "lang0" does not add a lot to the picture.

The misclassification rates per number of leaves shows a classic set of rate curves, with the U-shape of the validation results looking strong.



This reinforces what we learned about the interpretability of a simple classification tree. The splits on forks_count are intuitive and match my expectations as noted above. Anyone looking at the model can quickly understand that either a low or modestly high forks_count on a repository is a good indicator of what to expect. Ultimately, the takeaway lesson that if you see that at least three different people have forked a repository, it probably has at least ten people watching it, and is thus "popular" in some sense, is a decent rule of thumb that is easy to remember. The corollary that a project with only zero or one fork is unlikely to be successful yet is also intuitive and easy to remember.

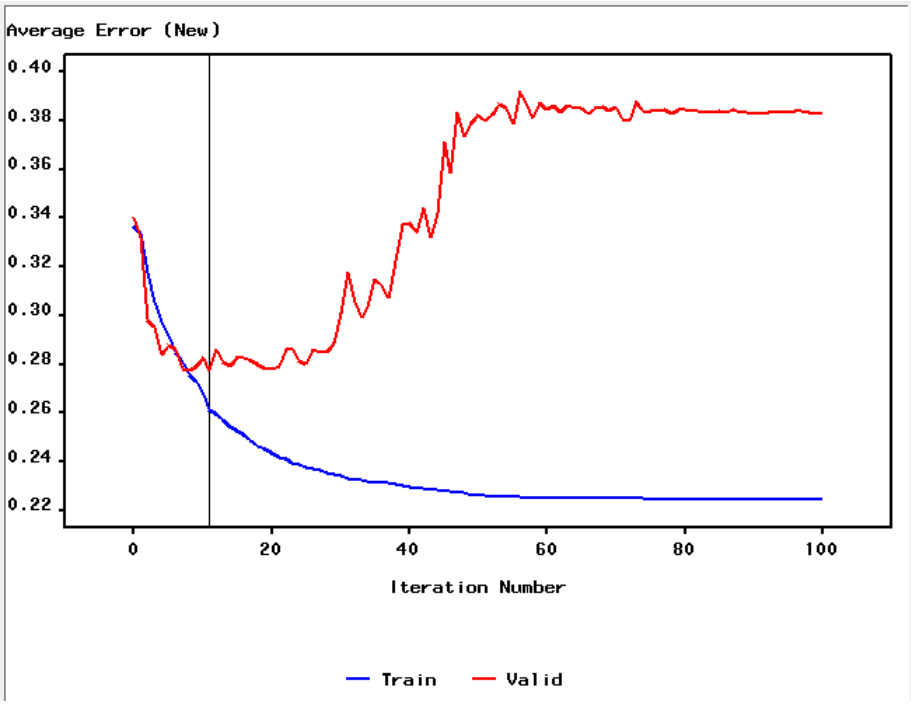
One final observation: the SAS tree model only found a handful of attributes even worth considering beyond forks_count and lang0, as indicated in this table:

Name	Importance	Role
FORKS_COUNT	1.0000	input
LANG0	0.2776	input
PUSHED_AT	0.1291	input
HAS_WIKI	0.1064	input
CREATED_AT	0.0916	input
ALL_COMMITS	0.0000	rejected
NETWORK_COUNT	0.0000	rejected
FORK	0.0000	rejected
OPEN_ISSUES_COUNT	0.0000	rejected
LANG1	0.0000	rejected
HAS_DOWNLOADS	0.0000	rejected
SIZE	0.0000	rejected
NUM_WEEKS	0.0000	rejected
HAS_ISSUES	0.0000	rejected
NUM_CONTRIBUTORS	0.0000	rejected
NUM_WEEKS_SINCE_CHANGE	0.0000	rejected
MEAN_COMMITS_PER_WEEK	0.0000	rejected
STD_COMMITS_PER_WEEK	0.0000	rejected
OWNER_COMMITS_PERCENTAGE	0.0000	rejected
LANG0_PROP	0.0000	rejected
LINES_SUBTRACTED_PER_WEEK	0.0000	rejected
LINES_SUBTRACTED	0.0000	rejected
LINES_ADDED_PER_WEEK	0.0000	rejected
LINES_ADDED	0.0000	rejected
OWNER_COMMITS	0.0000	rejected

Among these attributes, only lang0 from among all the language-related attributes, and only pushed_at from among the attributes indicating project activity, were deemed at all important. Even these were found to be of very low importance, on a par with has_wiki (discounted due to perceived unreliability in discussion above) and created_at, which might bring a component of bias for the older and oldest projects, which have simply had more time to be "starred", especially in the early days when very few repositories had even been registered yet.

5.2 Neural Network

For the neural network I used a Multilayer Perceptron with misclassification rate as the model selection criteria. Knowing that a neural network is at the opposite end of the interpretability spectrum, let's start with the average error chart:



The model performs fairly well after a few iterations but plateaus after around 10 and then degrades as we might expect after 30 or so iterations. A look at the cumulative lift chart reveals some improvements:

Here are the fit statistics for the neural network:

Fit Statistic	Training	Validation
[TARGET=star10]	.	.
Average Profit	0.1010301109	0.1010301109
Misclassification Rate	0.0724275987	0.0743390811
Average Error	0.2615204769	0.2773336243
Average Squared Error	0.0679808194	0.0727887766
Sum of Squared Errors	1096.7622953	295.41994657
Root Average Squared Error	0.2607313165	0.2697939522
Root Final Prediction Error	0.2675100434	.
Root Mean Squared Error	0.2641424263	0.2697939522
Error Function	4219.2165536	1125.584029
Mean Squared Error	0.0697712214	0.0727887766
Maximum Absolute Error	0.9974065266	0.9951778776
Final Prediction Error	0.0715616233	.
Divisor for ASE	16133.408	4058.592
Model Degrees of Freedom	207	.
Degrees of Freedom for Error	7859.704	.
Total Degrees of Freedom	8066.704	.
Sum of Frequencies	8066.704	2029.296
Sum Case Weights * Frequencies	16133.408	4058.592
Akaike's Information Criterion	4633.2165536	.
Schwarz's Baysian Criterion	6081.2851056	.

We see that the RMSE increases from 0.264 (training) to 0.27 (validation), a modest change that indicates that the model might do a decent job with predicting. The misclassification rate goes from 0.072 (training) to 0.074 (validation), also a modest change.

5.3 Regression

The regression model zeroed in on forks_count as well, and also pulled in "created_at" with a very low weight, which might just be an indication of popularity bias toward older projects in the sample, as noted above. Otherwise it is not an overwhelmingly compelling model. Here are the basic parameter estimates:

Effect Name	Effect Label	Parameter Estimate	Effect T-scores
Intercept	Intercept:star10=TRUE	40.620775893	6.2758363417
all_commits	all_commits	.	.
created_at	created_at	-0.003762697	-8.995166185
fork0	fork 0	.	.
forks_count	forks_count	0.8157817042	16.184934521
has_downloads0	has_downloads 0	.	.
has_issues0	has_issues 0	.	.
has_wiki0	has_wiki 0	.	.
lang0ACTIONSCRIPT	lang0 ACTIONSCRIPT	.	.
lang0ARC	lang0 ARC	.	.
lang0ASP	lang0 ASP	.	.
lang0ASSEMBLY	lang0 ASSEMBLY	.	.
lang0BRO	lang0 BRO	.	.
lang0C	lang0 C	.	.
lang0CLOJURE	lang0 CLOJURE	.	.
lang0COFFEESCRIPT	lang0 COFFEESCRIPT	.	.
lang0COLDFUSION	lang0 COLDFUSION	.	.
lang0COMMON_LISP	lang0 COMMON LISP	.	.
lang0CSS	lang0 CSS		

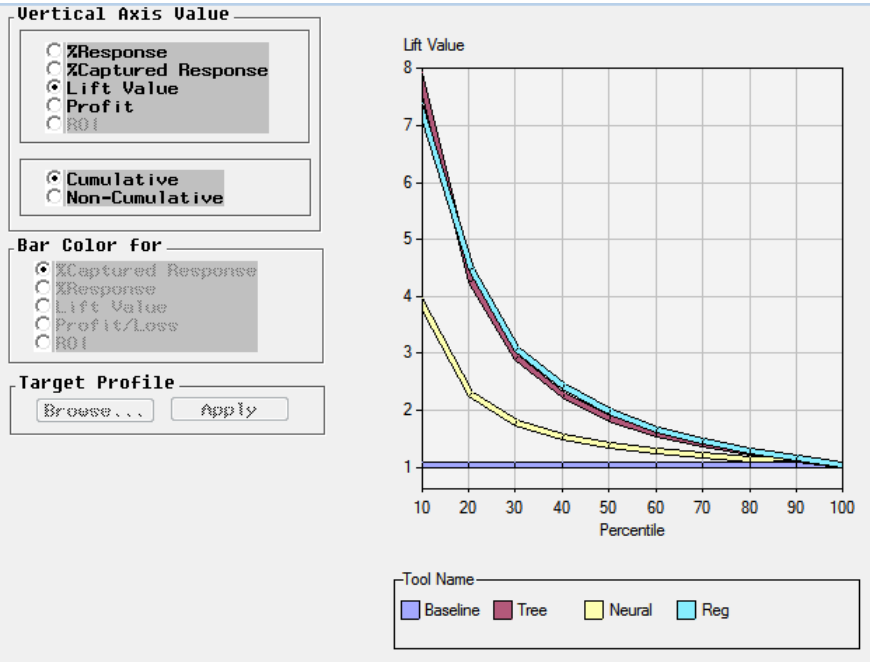
The performance of the regression model reveals a little more:

Fit Statistic	Label	Training	Validation
AIC	Akaike's Information Criterion	2251.4837425	.
ASE	Average Squared Error	0.0367658685	0.0440836235
AVERR	Average Error Function	0.1390582661	0.2342312616
DFE	Degrees of Freedom for Error	8062.704	.
DFM	Model Degrees of Freedom	4	.
DFT	Total Degrees of Freedom	8066.704	.
DIV	Divisor for ASE	16133.408	4058.592
ERR	Error Function	2243.4837425	950.64912452
FPE	Final Prediction Error	0.0368023485	.
MAX	Maximum Absolute Error	0.9917569925	0.9999999933
MSE	Mean Square Error	0.0367841085	0.0440836235
NOBS	Sum of Frequencies	8066.704	2029.296
_NW_	Number of Estimate Weights	4	.
RASE	Root Average Sum of Squares	0.191744279	0.2099610048
RFPE	Root Final Prediction Error	0.191839382	.
RMSE	Root Mean Squared Error	0.1917918364	0.2099610048
SBC	Schwarz's Bayesian Criterion	2279.4657435	.
SSE	Sum of Squared Errors	593.1587577	178.91744184
SUMW	Sum of Case Weights Times Freq	16133.408	4058.592
MISC	Misclassification Rate	0.0464417685	0.0536284505
PROF	Total Profit for STAR10	814.98	205.02
APROF	Average Profit for STAR10	0.1010301109	0.1010301109

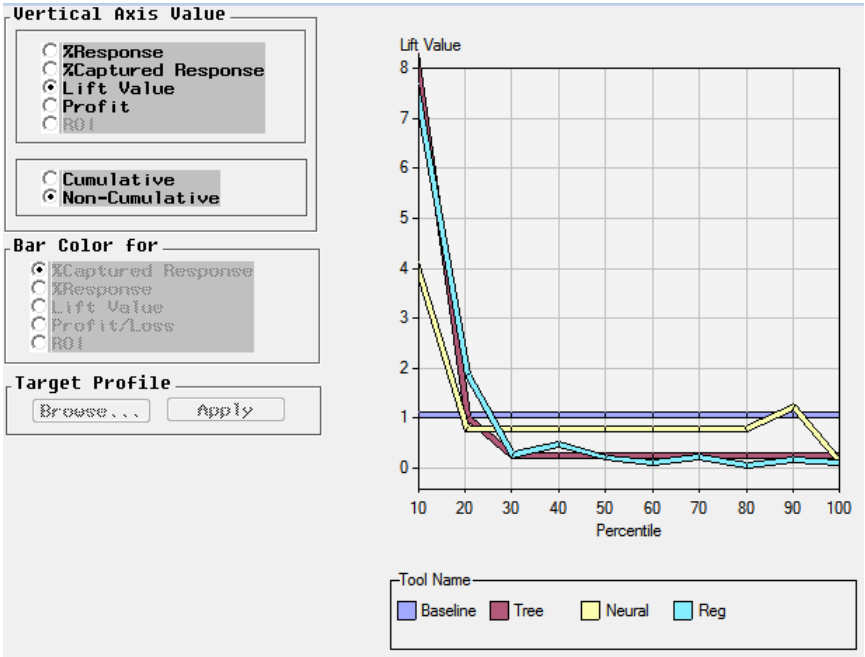
Here we see a RMSE of 0.192 in training and 0.21 in validation, indicating the model might be performing consistently. The misclassification rate increases slightly from 0.046 in training to 0.054 in validation as well.

5.4 Comparison

A look at the combined lift charts reveals no clear winner but a definite loser: the regression and tree models work pretty well, but the network does not perform as well. Here is the combined chart of cumulative lift:



Here we see that the tree model performs best, with a lift close to 8 in the first decile; the regression model performs nearly as well, still above 7, and the neural network model offers up a modest improvement at a lift of just under 4. At the second decile and beyond the regression model appears to outperform the tree model slightly. A closer look at the noncumulative lift is below:



In this chart the slightly higher slope of the regression model to a positive lift at the 20% decile is evident over the more quickly degrading lift of the tree model, but they are both offer only marginal results beyond the 10% decile. An interesting hiccup is the gentle spike at 90% for the neural network; perhaps it has some capacity to correctly determine low-end "not popular" classifications, even if this is not useful.

We can compare the tree and network models another way, using their misclassification rates:

Tool	Name	Description	Target	Target	B	B	Root ASE	Valid:Root ASE	Te	S	Misclassification Ra	Valid:Misclassification Ra
Tree	TreeGini	Tree	STAR10	True			0.1752078626	0.2038922427			0.0351365316	0.0479969408
Neural Network	NN	NN	STAR10	True			0.2607313165	0.2697939522		**	0.0724275987	0.0743390811
Regression	REGMISCALC	Regression	STAR10	True			.	.		.	.	.

In this table we see that the Root ASE of the tree model degrades a bit from training to validation but remains better than the network model at 0.204. The network model comes out at 0.27, degrading less than the tree model in validation but offering slightly worse performance. This pattern is duplicated in the misclassification rates, with the tree model dropping from 0.035 in training to 0.048 in validation, still better than the network model, which drops a touch from 0.072 in training to 0.074 in validation.

Assembling the misclassification rates from the various two-way charts above, we see:

Model	Misclassification Rate (training)	Misclassification Rate (validation)
Tree	0.035	0.048
Neural Network	0.072	0.074
Regression	0.046	0.054

According to this measure, the tree model outperforms both of the other

models in both training and in validation.

5.5 Conclusion

This project has taught me a great deal if the results are not particularly compelling. Each of the models zeroes in on forks_count as the key predictor of project popularity, and this makes good intuitive sense. It is not groundbreaking in any way, except perhaps in what it didn't find: there does not seem to be strong predictive value in the main languages a project uses or in the recent history of a project's commit activity, at least not present in the combination of attributes I used to train the models described here.

The process of collecting data was incredibly valuable to me as a lesson in suitability of any data to a particular task. I has assumed that collecting data from Github's well-documented API would be simple, and that the data would be "clean", but I spent many weeks iterating over the process and reconsidering the measures I was computing. I even found a bug in Github's API! All told, however, if I had allowed myself more time to work with the data itself I would likely have found more attributes to derive for consideration in these models.

There are many likely flaws in my results, beyond my nascent and weak skills in data mining. The collected data only comprises records from the first three years of github; in the three years since its popularity has grown immensely and the variety of projects registered with Github repositories has grown enormously. If I were to repeat this study the first step I would take would be to repeat the data gathering process to include a wider "stripe" of data from the entire lifetime of Github. A second step I would take would be to more closely examine the attributes I derived surrounding the activity around a project. I am certain I left out a number of important considerations.

Two deeper avenues for exploring this data exist as well and I pursued neither. First, as suggested by Professor Prasad during my project talk in class, the data I collected gives no indication of when users "starred" the repositories; it just includes the current count of stargazers. It would be interesting to track this number over time for a number of repositories and compare growth rates, looking for correlations between activity and attention/popularity. Finally, I would like to consider the code inside each repository. Can we glean anything from the actual code used to write applications, beyond programming language choices? For example, in Python many projects use built-in Python libraries and common third-party support libraries for frequently-needed tasks. Might there be a correspondence between the use of certain support project dependencies and their popularity, especially over time? It would be fascinating to build a dashboard to indicate which support libraries are "gaining traction" in each of the different programming language communities and in each of the respective application sectors. This, however, would require a lot more data and a lot more time.

6 Appendix

All code from this project and the derived dataset used to train the three models is available in a Github repository at https://github.com/dchud/substar. In particular, the following scripts handle the tasks described:

Script	Task
github.com/dchud/substar/blob/master/fetch.py	Fetch all data about repositories
github.com/dchud/substar/blob/master/process.py	Process collected data into CSV file
github.com/dchud/substar/blob/master/combined-10000.txt	Extracted dataset, n=10,067

The code for fetch.py (on which I spent the bulk of my time on this project) follows:

```
#!/usr/bin/env python

"""
Load up data about a number of repositories from github using their
API.  Report on results.
"""

import argparse
import glob
import json
import logging
import logging.config
from math import ceil
import os
import re
import time

import requests

from settings import *

logging.config.fileConfig('logging.conf')
logger = logging.getLogger('fetch')

HEADERS = {'Authorization': 'token␣%s' % TOKEN}


def wait_buffer(req):
    """by default, wait this long between requests to follow github's
    rate limits."""
    reset_seconds = int(req.headers['x-ratelimit-reset']) - ceil(time.time())
    remaining = float(req_full_data.headers['x-ratelimit-remaining'])
    # pad it a little, just to have a friendly cushion
    buffer = 1.1 * reset_seconds / remaining
    # whenever the timer gets down close to a reset, add extra cushion
    # note also, buffer should never be negative
    if buffer < 0.1:
        buffer = 0.5
    logger.debug('wait:␣%s' % buffer)
    time.sleep(buffer)


def repo_api_request(owner, name, func, count=0):
```

```
44      """
45      Retry-able api requests; handle 202 responses with 1+-second delay
46      retries up to MAX_RETRIES times with linear backoff.  Ignore rate
47      limit; 1+ seconds should always be longer than wait_buffer().
48      """
49      logger.debug('func: %s' % func)
50      r = requests.get('https://api.github.com/repos/%s/%s/%s' % (owner,
51          name, func), headers=HEADERS)
52      wait_buffer(r)
53      if r.status_code == 200:
54          return r.json()
55      elif r.status_code == 202:
56          count += 1
57          logger.debug('202 Accepted (count %s)' % count)
58          # linear backoff: always wait at least one extra second per retry
59          time.sleep(1 * count)
60          if count <= MAX_RETRIES:
61              return repo_api_request(owner, name, func, count=count)
62      logging.error(r)
63      return None
64
65
66  def save_recs(recs, count):
67      # write out the file to disk as an indented json file
68      filename = 'data/recs-%s.json' % count
69      fp = open(filename, 'wb')
70      json.dump(recs, fp, indent=2)
71      fp.close()
72      logger.debug('SAVED: %s' % filename)
73      return filename
74
75
76  def next_url(url):
77      # bump the sinceid 9900 to cycle through all repos over time
78      # from oldest to most recent
79      base, equalsign, sinceid = url.partition('=')
80      bumped_sinceid = int(sinceid) + 9900
81      return '%s=%s' % (base, bumped_sinceid)
82
83
84  if __name__ == '__main__':
85      parser = argparse.ArgumentParser(description='fetch github repo data')
86      parser.add_argument('-a', '--append', action='store_true',
87              default=False, help='pick up where we left off')
88      args = parser.parse_args()
89
90      if args.append:
91          logger.debug('appending')
92          files = glob.glob('data/*')
93          oldest_file = max(files, key=os.path.getctime)
94          logger.debug('oldest_file: %s' % oldest_file)
95          old_data = json.load(open(oldest_file))
96          oldest_rec_id = int(old_data[-1]['id'])
97          logger.debug('oldest_rec_id: %s' % oldest_rec_id)
98          bumped_rec_id = oldest_rec_id + 9900
99          req_repos = requests.get(
100                 'https://api.github.com/repositories?since=%s' % bumped_rec_id)
```

```
101             oldest_filename = oldest_file.split('/')[-1]
102             oldest_count_id = re.findall(r'\d+', oldest_filename)[-1]
103             count = int(oldest_count_id)
104         else:
105             # get a list of repos, starting from 0
106             req_repos = requests.get('https://api.github.com/repositories',
107                 headers=HEADERS)
108             count = 1
109         recs = []
110         next_repos_url = next_url(req_repos.links['next']['url'])
111         repos = req_repos.json()
112         while count <= FETCH_LIMIT:
113             logger.debug('count: %s' % count)
114             for repo in repos:
115                 owner = repo['owner']['login']
116                 name = repo['name']
117                 logger.debug('REPO: %s/%s' % (owner, name))
118 
119                 # get full data
120                 # /repos/:owner/:repo
121                 req_full_data = requests.get(
122                         'https://api.github.com/repos/%s/%s' % (owner, name),
123                          headers=HEADERS)
124                 full_data = req_full_data.json()
125                 rec = {'owner': owner, 'name': name}
126                 for key in ['id', 'full_name', 'url', 'homepage', 'git_url',
127                         'stargazers_count', 'watchers_count', 'subscribers_count',
128                         'forks_count', 'size', 'fork', 'open_issues_count',
129                         'has_issues', 'has_wiki', 'has_downloads', 'pushed_at',
130                         'created_at', 'updated_at', 'network_count']:
131                     rec[key] = full_data.get(key, '')
132 
133                 # NOTE: if there's never been a push, then forget it, jump
134                 #       to the next repo
135                 if rec['pushed_at'] is None:
136                     logger.debug('EMPTY, move on')
137                     continue
138 
139                 parent_keys = ['id', 'fork', 'forks_count', 'stargazers_count',
140                         'watchers_count', 'open_issues_count']
141                 wait_buffer(req_full_data)
142 
143                 # get contributors
144                 # /repos/:owner/:repo/[stats/]contributors
145                 # note: user stats/contributors because plain contributors
146                 #       paginates, even though it's a lot more data and can 202
147                 logger.debug('func: contributors')
148                 r = requests.get('https://api.github.com/repos/%s/%s/contributors' %
149                         (owner, name), headers=HEADERS)
150                 wait_buffer(r)
151                 # secondary check for uninteresting repo
152                 if r.status_code == 204:
153                     logger.debug('204, move on')
154                     continue
155                 contributors = r.json()
156                 while r.links.has_key('next'):
157                     logger.debug('func: contributors')
```

```
158                 r = requests.get(r.links['next']['url'], headers=HEADERS)
159                 contributors.append(r.json())
160                 wait_buffer(r)
161             rec['contributors'] = contributors
162 
163             # get participation
164             # /repos/:owner/:repo/stats/participation
165             participation = repo_api_request(owner, name, 'stats/participation')
166             if participation:
167                 rec['participation'] = participation
168 
169             # get languages
170             # /repos/:owner/:repo/languages
171             languages = repo_api_request(owner, name, 'languages')
172             if languages:
173                 rec['languages'] = languages
174 
175             # get code frequency
176             # /repos/:owner/:repo/stats/code_frequency
177             code_frequency = repo_api_request(owner, name,
178                     'stats/code_frequency')
179             if code_frequency:
180                 rec['code_frequency'] = code_frequency
181 
182             # get teams
183             # /repos/:owner/:repo/teams
184             # NOTE: url pattern 404s across repos
185             #teams = repo_api_request(owner, name, 'teams')
186             #if teams:
187             #    rec['teams'] = teams
188 
189             # get hierarchy
190             if full_data['fork']:
191                 if full_data['parent']['id'] != full_data['id']:
192                     rec['parent'] = full_data['parent']
193                     for key in parent_keys:
194                         rec['parent_%s' % key] = rec['parent'].get(key, '')
195                 if full_data['source']['id'] != full_data['parent']['id']:
196                     rec['source'] = full_data['source']
197                     for key in parent_keys:
198                         rec['source_%s' % key] = rec['source'].get(key, '')
199 
200             recs.append(rec)
201             if len(recs) == 100:
202                 save_recs(recs, count)
203                 recs = []
204 
205             count += 1
206             logger.debug('count: %s' % count)
207             if count == FETCH_LIMIT:
208                 break
209         logger.debug('FETCH: %s' % next_repos_url)
210         req_repos = requests.get(next_repos_url)
211         next_repos_url = next_url(req_repos.links['next']['url'])
212         repos = req_repos.json()
213 
214 if recs:
```

```
215         save_recs(recs, count)
```